Initial Report

Team Name: Gossip-4
Team Members: Wladislaw Meixner, Robert Schambach
Module: Gossip

Introduction

We are team **Gossip-4** and intend on implementing the **Gossip** module. Our team is composed out of Wladislaw Meixner and Robert Schambach.
Wladislaw Meixner is well versed in multiple programming languages especially golang, TypeScript, Swift, Python and others. With multiple years of golang server and typescript frontend development experience in the ecommerce market. Additionally, Meixner has proficient experience in server maintenance, Docker, Kubernetes setup as well as other dev ops tasks.
Robert Schambach is an experienced software engineer who is currently working in the IT-Security domain. Schambach has a well-rounded technical background; he has executed many projects in many languages including Rust, Bash, Java, and Python. Further, Schambach is proficient in integration and deployment of software systems, using technologies such as Docker, Docker-Compose, and Kubernetes.

Project overview

In this section, we describe the sketch of our Gossip module. We initially discuss the internal structure of the module. Further, we state the license under which we distribute the software of this module.

Design

The proposed design implements the Brahms ID-Exchange gossip system with random peer sampling [13]. As such, our project structure can be split into five key submodules: Communication, Persistence, ID-Exchange, ID-Validation and ID-Sampling.
The Communication submodule defines all message types listed in the project

specification, with matching serialization and deserialization procedures. Furthermore, it exposes an abstraction layer that other submodules can use for reliable communication with peers and the bootstrapping server.
The Persistence submodule allows other submodules to persist data on the host in a structured manner. We use a key-value based storage system here to save information, e.g. peer lists and other state information between executions.
Further, the ID-Exchange submodule consists of the ID-Exchange Gossip implementation of the system. This component sets how peers mutually acquire IDs. By acquiring IDs using the ID-Exchange Gossip type, the Gossip module avoids partitions and star-like topologies. This submodule thereby sends its ID and asks for IDs from and to random peers selected by the ID-Sampling submodule. The submodule thereby realizes the sending and asking of the IDs using the Communication submodule.
Complementary to the ID-Exchange submodule, the ID-Sampling submodule ensures that nodes acquired via ID-Exchange will be selected based on min-wise independent permutation before being used as peers. The Sampling submodule will additionally ensure its sampled nodes are active by periodically sending ping messages. The Sampling submodule may use the Persistence submodule to save its state to improve startup speed.
Auxiliary to the existing security advantages of Brahms ID-Exchange and Sampling submodules, peers use an ID-Validation mechanism to ensure the trustworthiness of new peers. A Proof-of-Work mechanism enforces the validity of a given peer.

Licensing

We aim to offer our project to others as a reference to supply value beyond the scope of this university course. Therefore, it is our intention to allow for equally open access and distribution of the source code and all of its artifacts. As such, we intend to publish all results under the permissive MIT-0 license [19]. In contrast to the MIT [20] license, the MIT-0 omits the attribution clause. This omission allows for unlimited sublicensing. Furthermore, we have made sure to select libraries that are at least as permissive as the MIT licensing agreement to match our goals.

Implementation

In the following section, we will discuss the aspects relevant for the realization of the Gossip module. These aspects include the selected programming language, libraries, build system and supported platforms.

Programming Language

Rust was chosen as the primary programming language for this project. This particular language was chosen for its high flexibility, performance, and memory safety. Additionally, Rust supplies useful built-in tooling, such as Cargo and rustfmt [16]. Moreover, the team members are partially acquainted with Rust.

Supported Platforms

The Rust programming language supports a multitude of operating systems and instruction set architectures out-of-the-box [1]. Further, during implementation, we take care to use operating-system agnostic concepts, e.g. fixed Unix file paths. As such, cross compilation for many platforms is easily achieved.
Yet, as we implement and test the module on Unix-based platforms, we primarily support such platforms. We do not guarantee the software to function as intended on other platforms, e.g. on Windows operating systems.

Build

Rust offers its own build system and package manager, namely Cargo [17]. We use Cargo to build, manage dependencies and potentially cross compile our Gossip module. Depending on the final project structure, it may be required to split our project into multiple independent submodules, which in terms requires further build tooling. We intend to use a combination of Cargo workspaces and Cargo scripts. Cargo workspaces aid with building and organizing of multiple packages, library crates in rust terminology, in a single monorepo project [11]. Additionally, we intend to use Cargo Build-Scripts for minor tasks such as Protocol Buffer generation.

Libraries

As Gossip is a networking module, we require libraries suitable for networking.
To this end, we use tokio [2], an asynchronous runtime for Rust. This library enables our module to use network functions in an asynchronous manner, offering substantial performance benefits without compromising the application's reliability, flexibility, or simplicity.
Further, every peer using Gossip must maintain peer status information consisting of peer IDs and corresponding network addresses. To store and retrieve these data without adding

additional overhead and complexity to the module, we use the embedded key-value store sled [4]. We prefer Slide to other embedded key-value databases, as Slide offers a simple interface and allows for optimization using simple local methods, with minimal user input.
We encode the Gossip messages using Protocol Buffers [5]. Furthermore, we utilize Protocol Buffers in Rust using rust-protobuf [6]. We chose this encoding as it is simple to write, maintain, and use.
To counter common peer-to-peer network attacks such as eclipse and sybill, this module enforces Proof-of-Work-based peer identities by requiring peers to calculate a hash with certain properties [7]. As these operations are computationally intensive, we choose the performant libraries ring [8] and rand [9]. Ring contains optimized functions for computing a suitable hash, and rand allows for fast generation of pseudo-random numbers.
For error handling, we primarily use the Rust standard error library. To avoid writing boilerplate code for error handling, we use the thiserror [3] library.
We parse the user-specified Windows INI formatted configuration file for Gossip using the ini [10] library.

Quality Assurance

A structured software development workflow promotes high code quality [18]. As we host our code in GitLab, we decide to use GitLab flows [14]. We want to adopt a merge request based GitLab workflow to keep the team in sync and catch problems early. After the team approval, GitLab pipelines are used to test functionality and quality. For testing these software attributes, we intend to use two pipelines, namely: Lint-Build-Test and End-To-End pipelines [15].
The Lint-Build-Test pipeline, as the name suggests, performs basic linting, Cargo check, builds the module and runs Cargo tests. Additionally, we intend to use LLVM coverage tools [12] in this pipeline to generate test and documentation coverage reports. GitLab runs the pipeline on all open merge requests; this process must succeed before a given PR can be merged into the main branch.
The End-To-End pipeline will be added later during the development process to test the Gossip module in its entirety and thereby ensure basic functionality and interoperability with other modules. As this pipeline is assumed to be slow, we only intend to run E2E tests on the main branch.
To ensure consistent code style, we intend to use pre commit hooks in combination with Cargo Husky [21] allowing us to vent code style and linting issues before a commit can be pushed to git.

Documentation is the final quality assurance measure included in this project. Rustdoc, a rust native documentation generation tool, is selected for this task. This approach allows us to write documentation in tandem with source code and enables automated documentation coverage tests. A key advantage of including documentation in the source code is that documentation and implementation are continuously kept in sync.

Workload distribution

The segments of the Gossip module described in the Design section partition the module's workload. We expect the submodules Peer Sampling and ID-Exchange to be the most sophisticated, as they require concepts introduced in the lecture [13]. We estimate the remaining submodules to be easier to implement, as they consist of commonly used logic. As such, we have distributed the workload partitions as follows:

Schambach's workload:

- Peer Sampling
- Persistence

Meixner's workload:

- ID-Exchange
- ID-Validation
- Communication

References

[1] "Platform Support - The rustc book." https://doc.rust-lang.org/nightly/rustc/platform-support.html (accessed May 28, 2022).
[2] "Tokio - An asynchronous Rust runtime." https://tokio.rs/ (accessed May 28, 2022).
[3] "thiserror - Rust" https://docs.rs/thiserror (accessed May 28, 2022).
[4] "sled - Rust." https://docs.rs/sled/ (accessed May 28, 2022).
[5] "Protocol Buffers | Google Developers." https://developers.google.com/protocol-buffers (accessed May 28, 2022).
[6] S. Koltsov, rust-protobuf. 2022. Accessed: May 28, 2022. [Online]. Available: https://github.com/stepancheg/rust-protobuf
[7] G. Carle, "Attacks in P2P Systems"

https://www.moodle.tum.de/pluginfile.php/3711620/mod_resource/content/1/05_P2P_Attacks.pdf (accessed May 19, 2022).
[8] B. Smith, ring. 2022. Accessed: May 28, 2022. [Online]. Available: https://github.com/briansmith/ring
[9] Rand. rust-random, 2022. Accessed: May 28, 2022. [Online]. Available: https://github.com/rust-random/rand
[10] “ini - Rust.” https://docs.rs/ini/1.3.0/ini/ (accessed May 28, 2022).
[11] “Cargo Workspaces” https://doc.rust-lang.org/book/ch14-03-Cargo-workspaces.html (accessed May 29, 2022)
[12] “Instrumentation Coverage” https://doc.rust-lang.org/nightly/rustc/instrument-coverage.html (accessed May 29, 2022)
[13] G. Carle, “Random Peer Sampling” https://www.moodle.tum.de/pluginfile.php/3715349/mod_resource/content/1/06_RPS.pdf (accessed May 19, 2022).
[14] “Gitlab Flow” https://docs.gitlab.com/ee/topics/gitlab_flow.html (accessed May 30, 2022)
[15] “CI/CD Pipelines” https://docs.gitlab.com/ee/ci/pipelines/ (accessed May 30, 2022)
[16] “Rust - Foreword” https://doc.rust-lang.org/book/foreword.html (accessed May 30, 2022)
[17] “Cargo” https://doc.rust-lang.org/Cargo/reference/specifying-dependencies.html (accessed May 30, 2022)
[18] “State of DevOps Report 2021” https://cloud.google.com/devops/state-of-devops (accessed May 30, 2022)
[19] “MIT-0 License” https://github.com/aws/mit-0 (accessed May 30, 2022)
[20] “MIT License” https://github.com/git/git-scm.com/blob/main/MIT-LICENSE.txt (accessed May 30, 2022)
[21] “Cargo Husky” https://github.com/rhysd/cargo-husky (accessed May 30, 2022)